AFRL-RI-RS-TR-2008-294
Final Technical Report
November 2008

ON LARGE-SCALE HYBRID COMPUTING ARCHITECTURE FOR NEOCORTICAL MODELS - WITH AN APPLICATION IN REALIZING COGNIZANCE OPERATIONS OF THE VISUAL CORTEX

State University of New York at Binghamton

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK

NOTICE AND SIGNATURE PAGE

Using Government drawings, specifications, or other data included in this document for any purpose other than Government procurement does not in any way obligate the U.S. Government. The fact that the Government formulated or supplied the drawings, specifications, or other data does not license the holder or any other person or corporation; or convey any rights or permission to manufacture, use, or sell any patented invention that may relate to them.

This report was cleared for public release by the 88th ABW, Wright-Patterson AFB Public Affairs Office and is available to the general public, including foreign nationals. Copies may be obtained from the Defense Technical Information Center (DTIC) (http://www.dtic.mil).

AFRL-RI-RS-TR-2008-294 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

FOR THE DIRECTOR:

/s/	/s/
THOMAS E. RENZ Work Unit Manager	EDWARD JONES, Deputy Chief Advanced Computing Division Information Directorate

This report is published in the interest of scientific and technical information exchange, and its publication does not constitute the Government's approval or disapproval of its ideas or findings.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
NOV 2008	Final	Jul 07 – May 08

4. TITLE AND SUBTITLE

ON LARGE-SCALE HYBRID COMPUTING ARCHITECTURE FOR NEOCORTICAL MODELS – WITH AN APPLICATION IN REALIZING COGNIZANCE OPERATIONS OF THE VISUAL CORTEX

5a. CONTRACT NUMBER

5b. GRANT NUMBER
FA8750-07-1-0195

5c. PROGRAM ELEMENT NUMBER
61101E

6. AUTHOR(S)

Qing Wu and Qinru Qiu

5d. PROJECT NUMBER
459T

5e. TASK NUMBER
AC

5f. WORK UNIT NUMBER
GR

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
State University of New York at Binghamton
Dept of Electrical and Computer Engineering
PO Box 6000
Binghamton NY 13902

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

AFRL/RITC
525 Brooks Rd.
Rome NY 13441-4505

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSORING/MONITORING AGENCY REPORT NUMBER
AFRL-RI-RS-TR-2008-294

12. DISTRIBUTION AVAILABILITY STATEMENT
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED. PA# 88ABW 2008-0962

13. SUPPLEMENTARY NOTES

14. ABSTRACT

This report describes working hardware and software developed to realize large-scale Brain-State-in-a-Box (BSB) models on a workstation with hardware acceleration using a Field Programmable Gate Array (FPGA). Just one Xilinx XC2VP70 FPGA was able to support about 600 128-dimensional BSB models to run at 10ms reaction time. Software was developed that controls the hardware operations and sends/receives data through publish/subscribe routines provided by an open-source package. Next, the confabulation based knowledge base training function on the Cell Broadband Engine (CBE) was implemented. The workload of the training function was distributed to 6 Synergistic Processing Elements (SPEs) in the Cell processor. Dynamic memory management techniques were developed to enable the SPE to load and write back information from/to the main memory during the training process. Preliminary software profiling was performed to indicate the performance bottleneck and guide the software optimization. The Cell-based implementation achieved 4X~9X speedups comparing to traditional processors.

15. SUBJECT TERMS
Brain State in a Box, Hardware for Cognitive Systems, Hardware Cognitive Operations, Modular Cognitive Computer Systems

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			Thomas Renz
U	U	U	UU	32	19b. TELEPHONE NUMBER (*Include area code*) N/A

Table of Contents

List of Figures

List of Tables

1. SUMMARY

This research project consists of two major tasks.

In the first task we have investigated and developed a novel high-performance *bio-inspired computing architecture* (BICA) for realizing the cognizance operations of the human brain. We developed working hardware and software that realize large-scale Brain-State-in-a-Box (BSB) models on a Windows-based workstation with hardware acceleration using Field Programmable Gate Array (FPGA).

The hardware versions of the 128-dimensional BSB training and recall functions were implemented and run on the Annapolis Wildcard II Pro FPGA board in a Dell Precision workstation. With just one Xilinx XC2VP70 FPGA, we were able to support about 600 128-dimensional BSB models to run at 10ms reaction time. We also developed software that controls the hardware operations and sends/receives data through publish/subscribe routines provided by an open-source package.

In the second task, we implemented the confabulation based knowledge base training function on the Cell Broadband Engine. The workload of the training function was distributed to 6 Synergistic Processing Elements, SPEs in the cell processor. We developed dynamic memory management techniques to enable the SPE to load and write back information from/to the main memory during the training process. Preliminary software profiling was performed to indicate the performance bottleneck and guide the software optimization. Compared to single processor training function that runs on a workstation with dual core 2GHz Pentium processor, the cell based implementation achieves 4X~9X speedups.

2. INTRODUCTION

2.1. Human cognizance and neo-cortical models

Nerve cells, called neurons are the fundamental components of the central nervous system (or the brain). The brain consists of approximately 100 billion neurons (10^{11} neurons). Neurons have 5 special functions which differ from the other cells in the body: (1) they receive signal pulses from their neighboring neurons; (2) they integrate these input pulses; (3) sufficient input pulses give rise to output pulses; (4) neurons can conduct these pulses; (5) they transmit them to other neurons which are capable of receiving them.

As shown in Figure 1, the neuron is made of three parts: cell body, dendrites and axon.

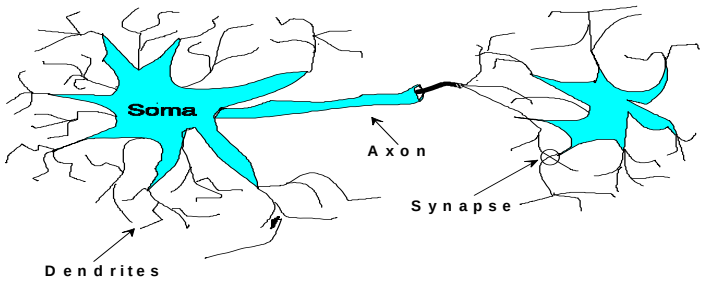


Figure 1. Components of a neuron.

The body of the cell contains the nucleus of the neuron and carries out the various biological transformations necessary to the life of neuron. The dendrites are the principal receptors that receive the incoming signals from the other neurons. The axon or nerve fiber sends out the output signals from the neuron. Axons branch out to communicate with other neurons. Neurons are connected to each other in a complex spatial arrangement to form the central nervous system. As shown in Figure 1, the connection between two neurons takes place at the synapse, where they are separated by a synaptic gap of the order of 0.01 micron.

The neuron processes the electric signals that arrive on dendrites and transmits the resulting electric signals to other neurons through the axon. The classical biological explanation of this processing is that the cell carries out a summation of the incoming signals on its dendrites. If the summation exceeds a certain threshold, the neuron responds by issuing a new pulse that is propagated along its axon. The neuron remains in an inactive state if summation is less than the threshold. This model is shown in Figure 2.

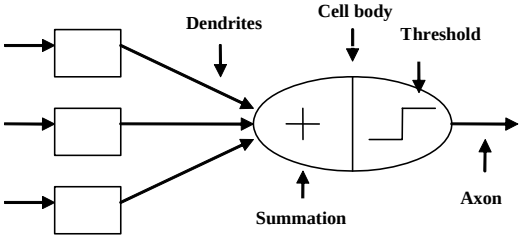


Figure 2. The neuron model.

Research shows that, the human neocortex is divided into four main lobes. The lobes are divided into cortical columns. Cortical columns are further divided into mini columns that are made up of about 100 to 200 neurons as shown in Figure 3.

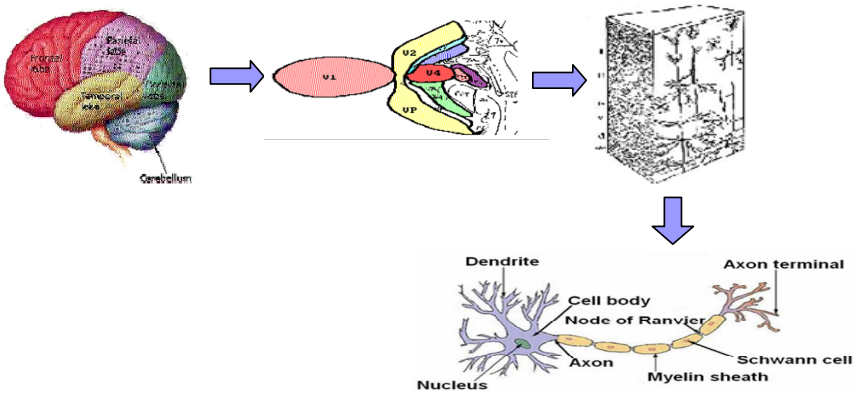


Figure 3. Structure of the human brain.

With the recent ongoing research in the field of human cognizance, it has been shown that the working mechanisms of the auto-associative and hetero-associative neural memory models are very similar to that of the *cerebral cortex* i.e., *neocortex*. Researchers think that the neocortex follows a hierarchical architecture [2] as shown in Figure 4. On the bottom of the hierarchy is the neuron; multiple neurons forming cortical mini-columns; multiple mini-columns forming cortical columns, with the pattern repeated at higher levels to implement the functional blocks thought to underlie cognizance operations, for example vision, in the human brain.

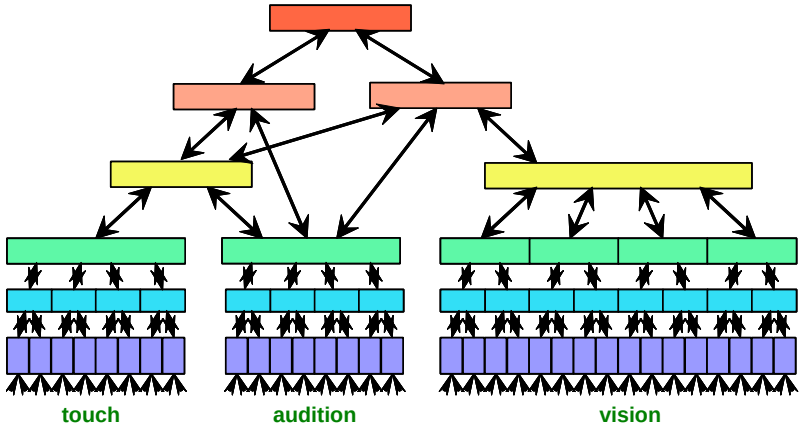


Figure 4. A hierarchical model of the neocortex.

Various mathematical models have been studied to mimic the different operations in this hierarchical architecture/functionality of the brain. The Brain-State-in-a-Box (BSB) attractor model, is one of the promising solutions to the problem. The BSB model is usually used to model the functionality of a mini-column. Multiple BSB models can be connected to model a cortical column, and eventually to model a complete cognitive function of the brain such as vision.

2.2. Brain-State-in-a-Box (BSB) model

To artificially realize the operations in this hierarchical architecture/functionality of the brain, different mathematical models have been studied. The BSB attractor model [1] is one of the promising solutions to the problem. The BSB model is a simple, auto associative, nonlinear, energy-minimizing neural network. There are two main operations in a BSB model, *Training* and *Recall*.

The mathematical model of a BSB recall operation can be written as:

$$\mathbf{X}(t+1) = S(\alpha \cdot \mathbf{A} \cdot \mathbf{X}(t) + \lambda \cdot \mathbf{X}(t) + \gamma \cdot \mathbf{X}(0))$$

where:

- $\mathbf{X}(t)$ and $\mathbf{X}(t+1)$ are N dimensional vectors;
- $\mathbf{A}$ is an $N \times N$ connection matrix;
- α is a scalar constant feedback factor;
- λ is an inhibition decay constant;
- γ is a nonzero constant if there is a need to maintain the input stimulation;
- $S()$ is the "squash" function: $S(x) = 1$ if $x > 1$; -1 if $x < -1$; x otherwise;

In the training operation, the $N \times N$ connection (weight) matrix $\mathbf{A}$ is modified as

$$\Delta\mathbf{A} = lr * (\mathbf{X} - \mathbf{AX}) \otimes \mathbf{X}$$

$$\mathbf{A} = \mathbf{A} + \Delta\mathbf{A}$$

where, $\mathbf{X}$ is the normalized input training pattern;

lr is the Learning rate;

$(\mathbf{X} - \mathbf{AX}) \otimes \mathbf{X}$ is the outer product of two vectors;

2.3. The hybrid computer cluster platform

The proposed hardware architecture is targeted at highly-connected hybrid computer clusters, which consist of a large number of workstations communicating with each other through high-speed interconnect networks. Within each workstation, in addition to traditional architecture with general-purpose processors, there are custom boards with field programmable gate array, FPGA devices and local memories.

Figure 5 shows the components and system structure of the High-Performance Computing, HPC cluster at the Air Force Research Lab, Rome, New York. The HPC cluster consists of about 50 computing nodes that are connected through a high-speed interconnect network. Each node in the cluster consists of a general-purpose workstation with Intel's Pentium Xeon processors running the Linux operating system, and a WILDSTAR II PCI card [3] in the workstation's PCI slot.

Figure 6 shows the detailed block diagram of the WILDSTAR II PCI card. There are two Xilinx Virtex II XC2V6000 FPGA [4][5] Processing Elements, PEs on each card. Each PE connects to 6 parallel local memory banks, which provides high bandwidth (5.5 GBytes/second) for data read/write operations. These high-performance FPGA cards are the key enabling technology for the proposed computing architecture.

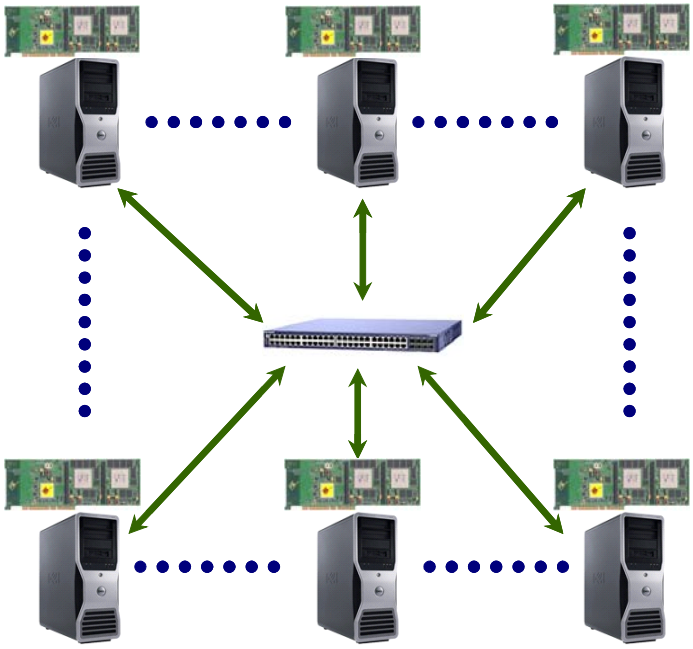


Figure 5. The components and system structure of the HPC cluster at RomeLab.

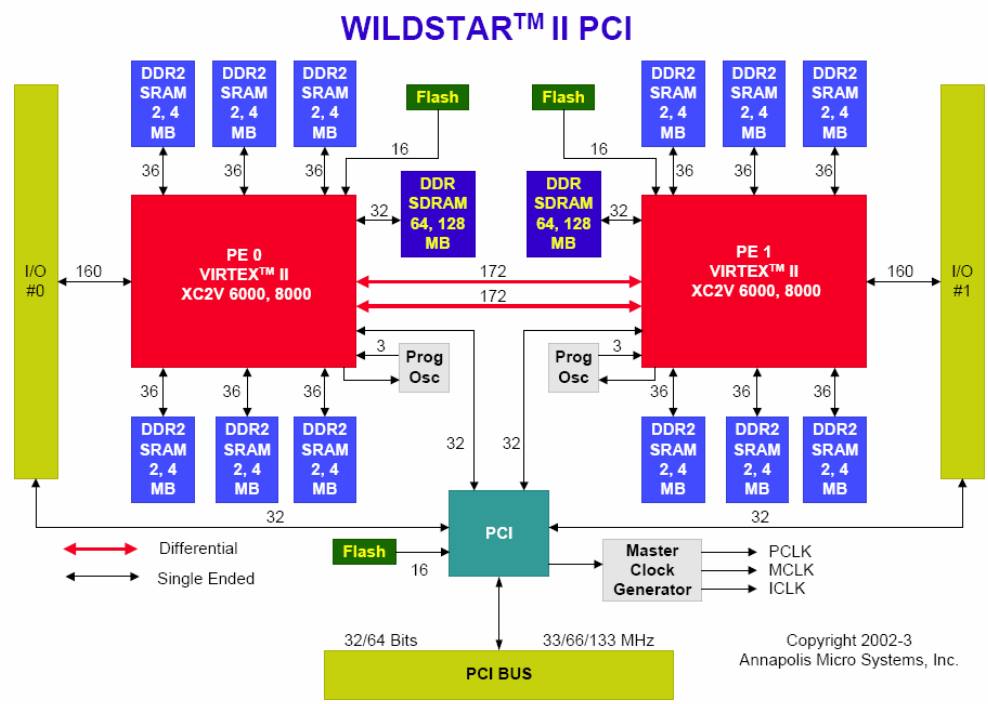


Figure 6. The block diagram of the WILDSTAR II PCI card.

2.4. Confabulation based knowledge base training

Cogent confabulation is an emerging theory proposed by Dr. R Hecht-Nielsen. Based on the theory, the information processing of human cognition is carried out by thousands of separate thalamocortical modules. Each of these thalamocortical modules is a patch of cerebral cortex plus a uniquely paired zone of thalamus and is referred to as a *lexicon* or a *feature attractor module*. Different collections of neurons in the thalamocortical module represent different symbols. Knowledge is stored as the links and their strength between neurons.

Confabulation based sentence completion software has been developed at the Air Force Research Laboratory Emerging Computing Technology Branch, AFRL/RITC, based on the example provided in [7]. (Two versions of the confabulation software have been developed. The code "cf.c" was developed by Daniel Burns from AFRL/RITC while the codes ""LexMaker.cc" and "TextReader.cc" were developed by Michael Moore from ITT Industries.) A sentence is represented using 40 lexicons that are arranged in 2 levels. A lexicon is a collection of symbols. The *i*th lexicon in level 1 represents the word (or punctuation) in the *i*th location of a sentence. Since there are 20 lexicons in level 1, the words or punctuations beyond the first 20 are discarded. The *i*th lexicon in level 2 represents the phrase that initiates from the *i*th location of a sentence. The connections of knowledge bases are "causal". Within each level, each lexicon only establishes knowledge bases with later lexicons. The lexicons in different levels are also connected with knowledge bases. The *i*th lexicon in level 1 is connected to the *j*th lexicon in level 2 where $j \le i$ while the *j*th lexicon in level 2 is connected to the *i*th lexicon in level 1 where $i \ge j$. shows the lexicons and the knowledge bases for the sentence completion problem.

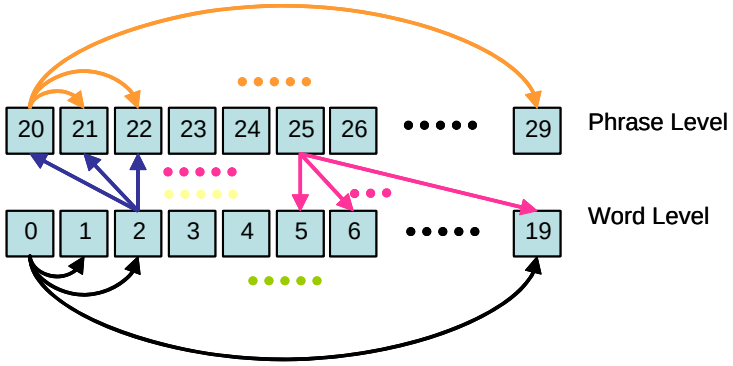


Figure 7. Lexicons and knowledge bases for confabulation based sentence completion.

Overall, there are 800 knowledge bases in the system. A knowledge base from lexicon A to B is a matrix. Each row in the matrix represents a source symbol that appears in A. Each column represents a target symbol that appears in B. The ijth entry in the matrix represents the strength of the link between symbol *i* and *j*. It gives the number of times that symbol *i* and *j* co-occurs in lexicon A and B respectively. The knowledge base matrices are built up during the learning process. They are learned by reading novels and scientific papers that are stored on hard disk.

The size of the knowledge bases depends on the training files. Let *N* denote the total number of possible words, phrases and punctuations in the training file, then each lexicon has a collection of *N* symbols and each knowledge base is an $N \times N$ matrix. For the sentence completion problem, since the symbols are the words and phrases in the English dictionary, the size of *N* can easily go up to more than 10,000. It is almost impossible to store the entire matrix without compression.

Our previous analysis [8] shows that the merged tree structure is the most efficient for managing the knowledge base during the training process. Figure 8 gives an example of the merged tree based representation of the knowledge base. A source tree is associated with each source lexicon. Each node of the source tree represents a symbol that appears in the source lexicon during the training. A source lexicon has multiple (20~40) links pointing to different target lexicons. Hence, each node in the source tree points to a list of target trees. Each target tree stores the set of symbols that appear in the corresponding target lexicon. Overall, there will be 40 source trees and a large number of target trees.

The size of the source or target trees is determined by the training file. In our previous study, the system was trained using a medium sized training file. The average size of a source tree is 4000 nodes while the average size of a target tree is 3 nodes. Overall, there are 4000*40 = 160K source nodes and 160K*3 = 480K target nodes.

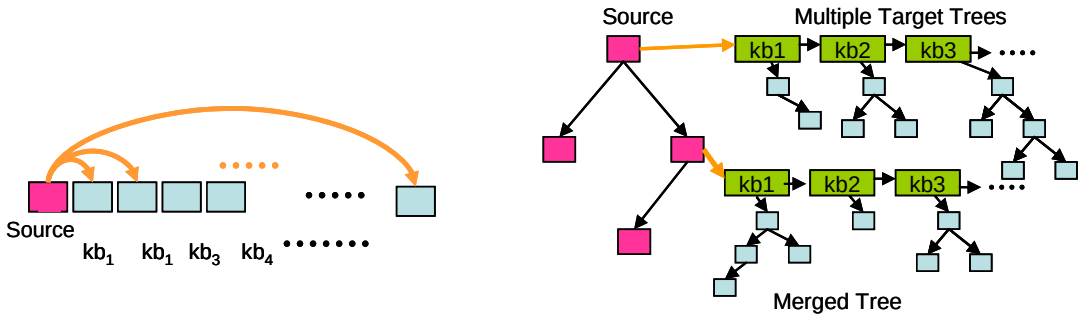


Figure 8. Representing the knowledge base using merged tree.

Each tree node in the source tree or target tree requires at least 4 integer fields. Both of them need to store the information of the symbol_id, the pointer to left and right children. The node that belongs to a source tree is also associated with a pointer that points to the list of target trees while the node that belongs to a target tree is associated with an integer which is the value of the knowledge base. Figure 9 shows the detailed data structure for one source tree and its target trees. The above analysis shows that each tree node requires 16 bytes storage.

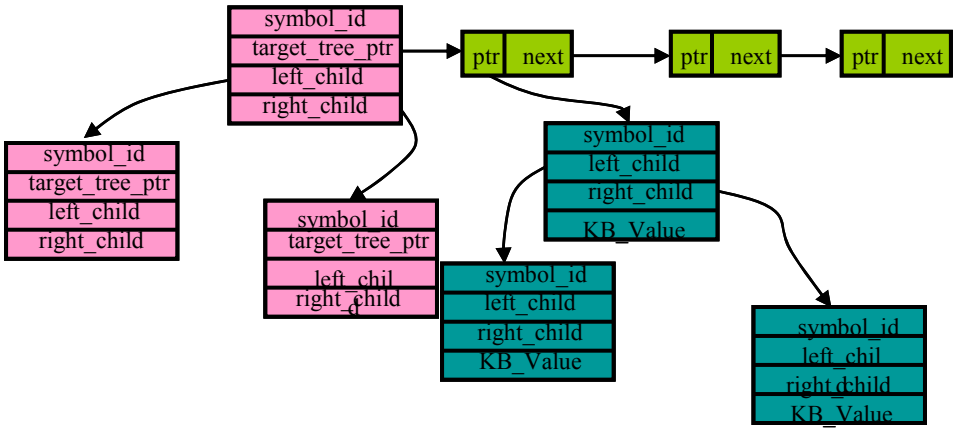


Figure 9. Detailed data structure for source and target trees.

2.5. Web solutions platform event system: a distributed publish/subscribe event system

The Web Solutions Platform [6], WSP event system is a distributed publish/subscribe event system. It is distributed in that the publishing and subscribing of events can be intra-machine and inter-machine. Publishers and subscribers do not know of each other's existence although a publisher could listen to subscription events to determine if there is an interested subscriber to its events prior to publishing.

Events have an Event Type property to define their type. This property is a Globally Unique Identifier, GUID which allows for anyone to define their own event type without worry of conflicting with event types created by other individuals. Applications subscribe to event types. The application can subscribe to specific event types or to all event types. Each subscription made is assigned a subscription ID. A subscription event is then published throughout the mesh for this subscription ID and event type.

The machines in the event mesh are organized in a hierarchy. Each machine knows who its parent is and the parent learns about its children as they connect. Communication between machines is done using Transmission Control Protocol, TCP. If events are sent between siblings, the events are sent from the publishing machine to the parent machine and then down to the subscribing machine. Regardless of how many subscribers there may be from the parent onward, the event will only be sent once from the child machine to the parent machine (or parent to child if that is the case). The parent will in turn route the event on to its parent and/or children as appropriate.

The WSP event system is composed of a collection of loosely-coupled components. The system runs as a windows service with each component running in its own thread. The components communicate with each other through in-memory queues.

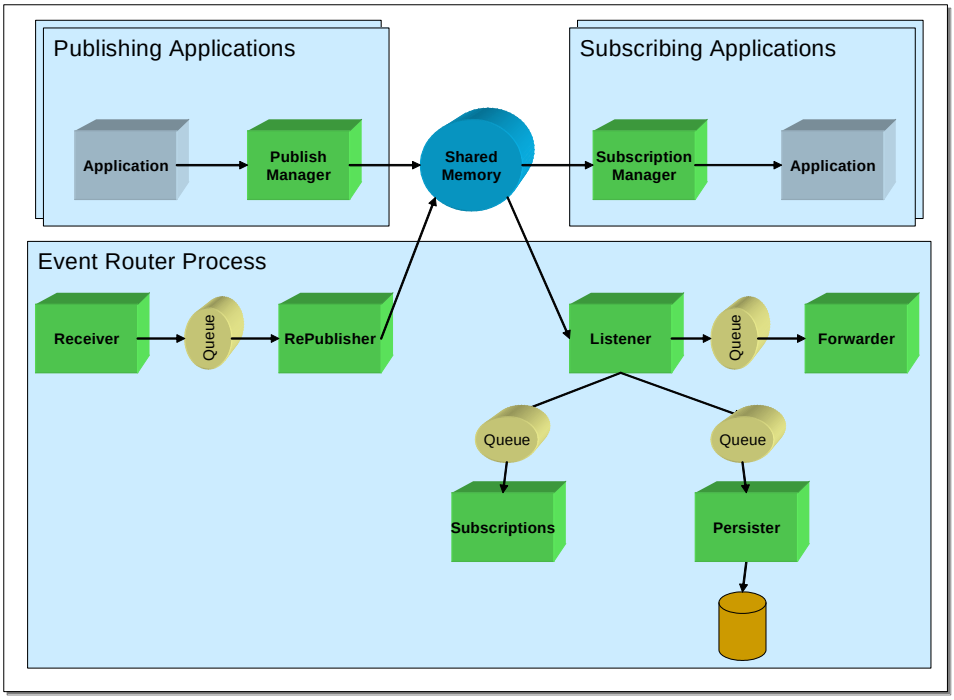


Figure 10. The block diagram of the WSP system.

Figure 10 shows the main components of the system. The Receiver and Forwarder are the communications interfaces to interact with the parent and children machines. When an event arrives to the Receiver, it

hands the event off to the RePublisher to publish the event on the local system. Publishing the event means putting it in the Shared Memory buffer. Interested subscribers copy the event from Shared Memory as does the Listener. The Listener forwards the event to the other components, if appropriate. If the event is a subscription event, the Listener sends it to Subscriptions to update the routing tables. If the event type is to be persisted, the Listener forwards the event to Persister. If the routing tables indicate the event needs to be forwarded, the Listener will forward the event to the Forwarder to route the event.

For publishing applications, they simply put the event into the Shared Memory buffer. When the publisher and subscriber are on the same machine, the event is placed into Shared Memory by the publisher and taken out of Shared Memory by the subscriber. It never incurs the overhead of going through a broker, e.g. the event system. Since there can be N subscribers listening to what events are placed into Shared Memory, this allows for all N subscribers to get the event at the same time and very efficiently.

3. METHODS, ASSUMPTIONS, AND PROCEDURES

3.1. Hardware designs of a single 128-dimensional model

The overall hardware/software system architecture is show in Figure 11.

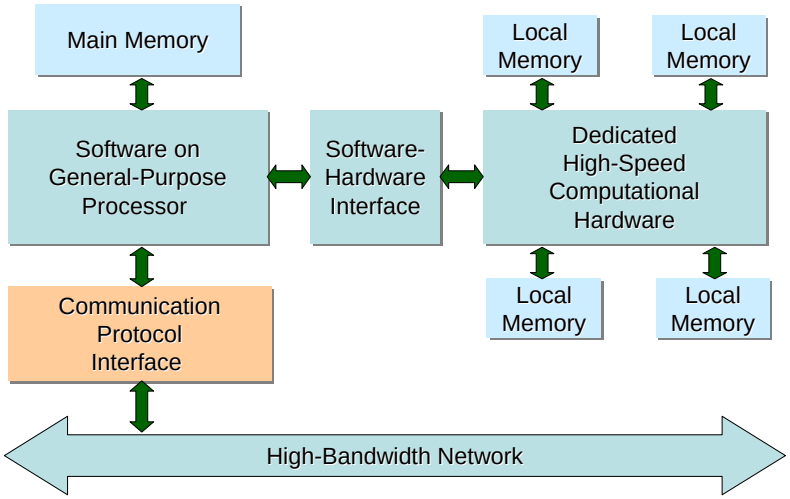


Figure 11. Overall system architecture.

Figure 12 shows the data path design of the hardware accelerator for 128-dimensional BSB recall operations. The 128×128 weight matrix is read from the interface BRAM in 32 stages and stored in 128 on-chip Block Random Access Memory, BRAMs. The BRAMs are selected one by one using the index counter. The **X** vector is read and stored in two 128-stage shift registers, of which one is used for parallel multiplication and the other for serial addition. The **X** vector is then multiplied with each row element of the weight matrix from 128 BRAMs and the values are latched to pipeline register P1. The values from P1 are added using the adder tree. An additional pipeline stage P2 is included inside the adder tree. The output of the adder tree is latched to the pipeline register P3. This value is multiplied with α to get $\alpha\mathbf{AX}$ which is then latched to the pipeline register P4.

The value $\lambda\mathbf{X}$ is calculated from the 128-stage **X** shift register used for serial addition. This value is delayed for 4 clock cycles to match the pipeline delay due to the other path using 4-stage pipeline register P5, which is then added with the value in the register P4. This value is squashed and the new value of **X** is

stored in 128-shift register $\mathbf{X}_N$ in 132 clock cycles. Finally, the value in $\mathbf{X}_N$ is updated to both the **X** shift registers in one clock cycle.

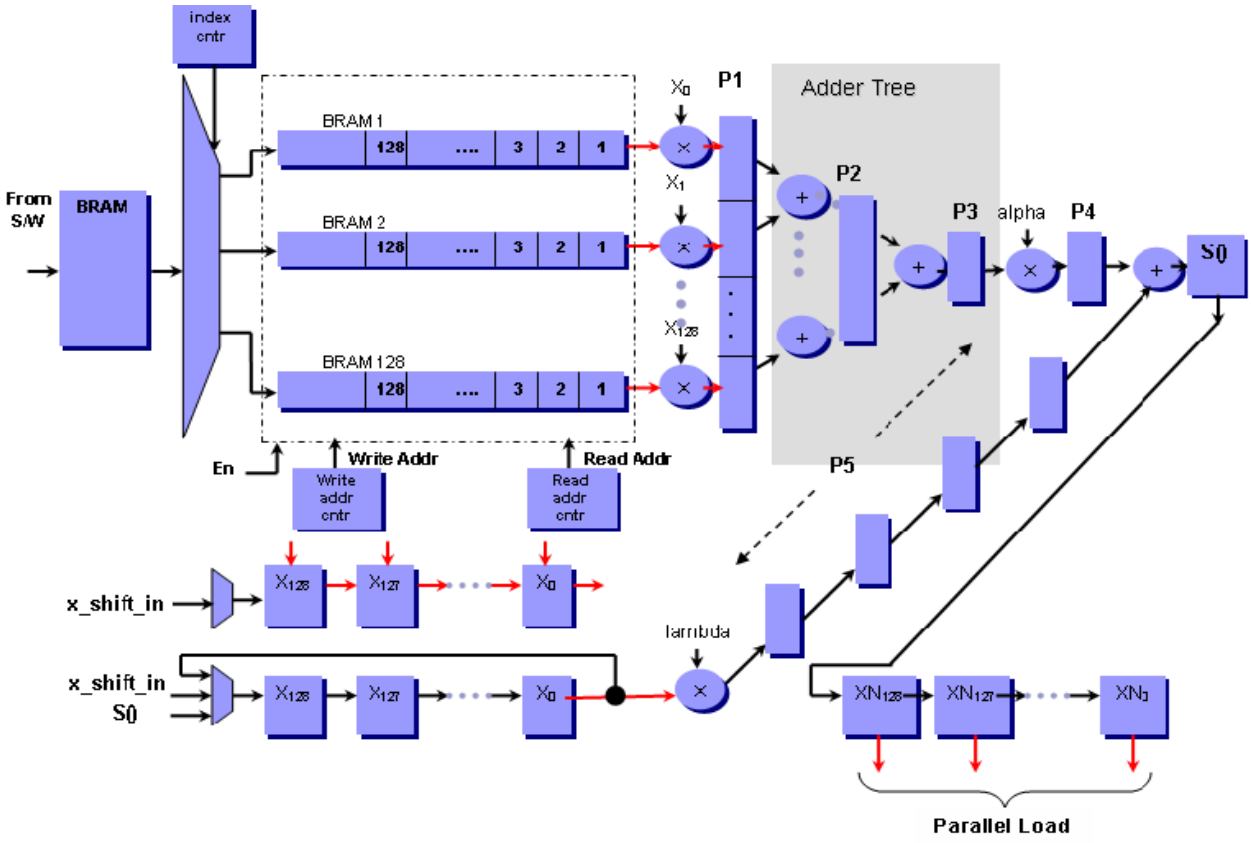


Figure 12. Data path design of the hardware accelerator for BSB recall operation.

Figure 13 shows the data path design of the hardware accelerator for 128-dimensional BSB training operations. The main goal of training is to generate a weight matrix from the given **X** vector. The normalized value of **X** vector ($\mathbf{X}_{norm}$) is read and stored in two 128-stage shift registers, of which one is used for parallel multiplication and the other for serial subtraction. The weight matrix is initialized to '0'. The $\mathbf{X}_{norm}$ is then multiplied with each row of the weight matrix in parallel and is latched in the pipeline register P1 every clock cycle. The adder tree performs the addition of all the elements in the pipeline register P1 and latches the result to pipeline register P3. There is an additional pipeline stage P2 inside the adder tree. The value in P3 is then subtracted from $\mathbf{X}_{norm}$ (this value is provided from the serial shift register). The result is multiplied by learning rate and then latched to the pipeline register P4. The resultant value $lr \cdot \mathbf{(X - AX)}$ is shifted into another serial shift register each clock cycle. Then an outer product operation is performed on X_{norm} with each value of $lr \cdot \mathbf{(X - AX)}$ from the serial shift register. The result is latched to the pipeline register P5. Each row value of the weight matrix is delayed 1 clock cycle to match the pipeline delay due to the other path and is added with the result in P5. The result is then updated to the corresponding row of the weight matrix. The 128 parallel multipliers are shared between

the first parallel multiplication with weight matrix and the second with outer product using a multiplexer and a de-multiplexer.

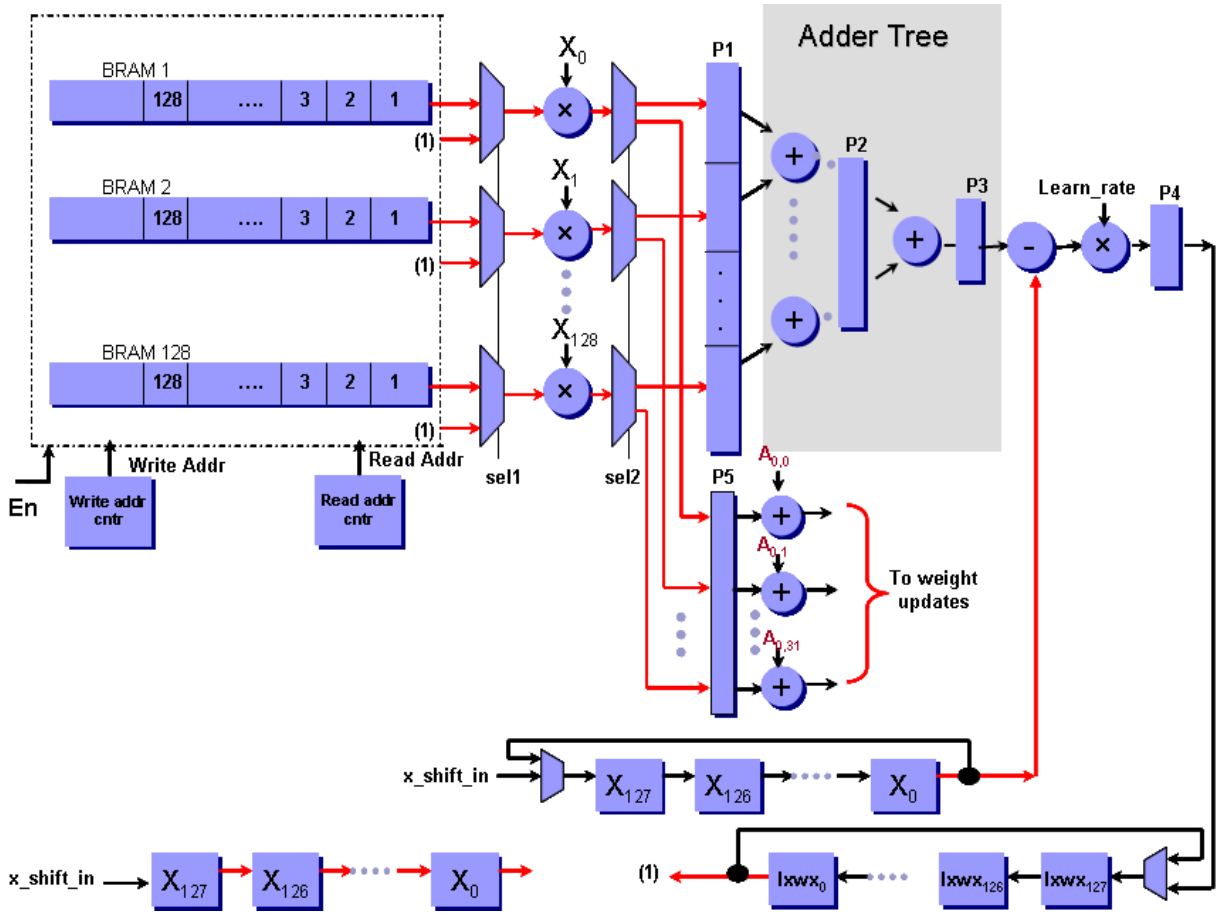


Figure 13. Data path design of the hardware accelerator for BSB training operation.

3.2. Hardware Design for running 256 128-neuron BSB models on the same FPGA

Due to the memory limitation of the FPGA board, the final design can accommodate 256 different weight matrices. This large amount of data can only be stored using SRAM chips on the board. The design has 6 parallel direct access channels to the 6 local SRAM banks, to minimize the loading time of the coefficients. The design is shown in Figure 14.

The weight matrices from the software are loaded into the SRAMs through the BRAM interface one by one. Each weight matrix is divided among the 6 SRAMs, for parallel load. This step is the same as mentioned in the previous design.

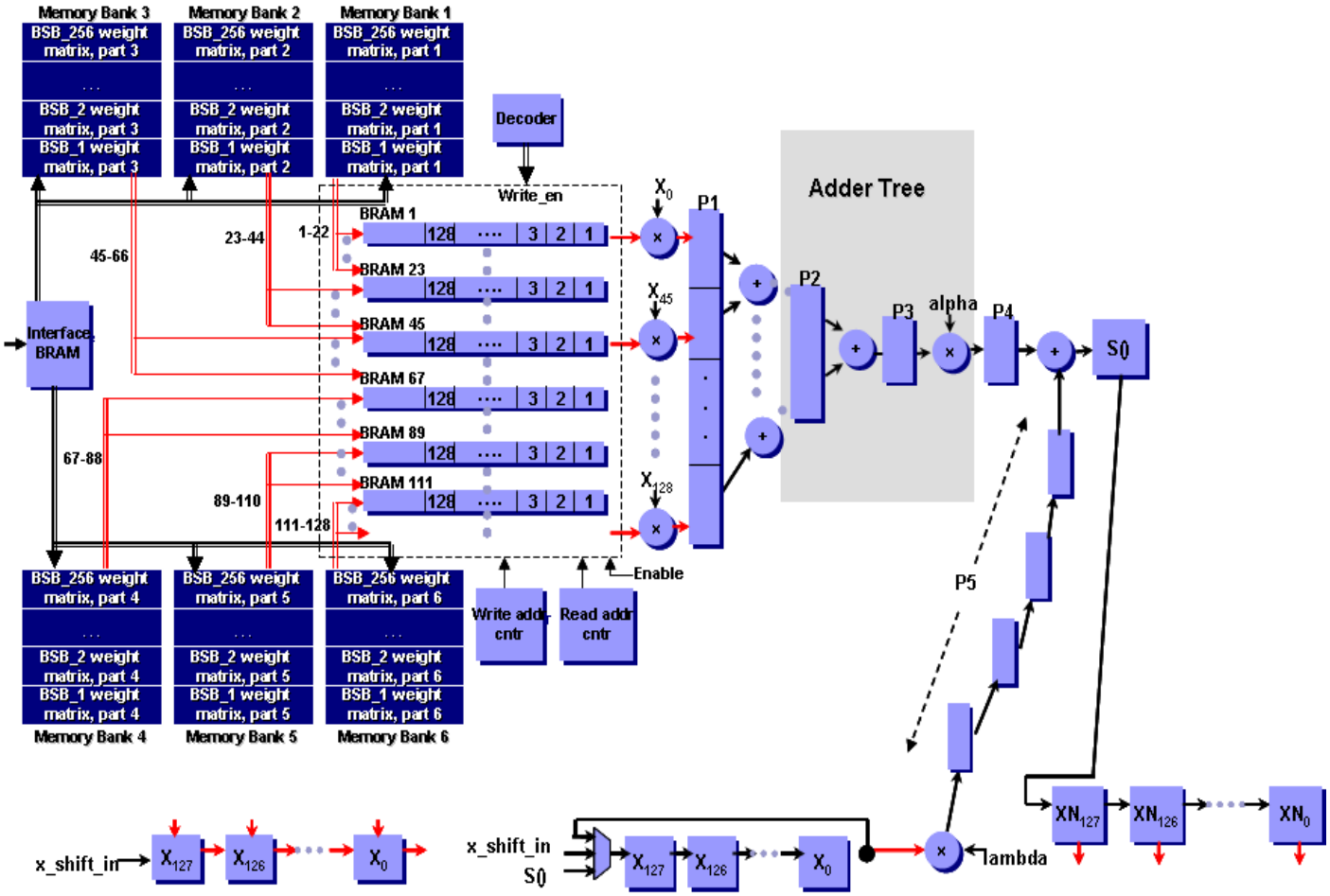


Figure 14. Data path design of the hardware accelerator for supporting 256 different BSB models.

The weight matrix is loaded from the 6 SRAMs into the 128 BRAMs in parallel (i.e. 6 BRAMs are loaded in parallel at a time). This task is synchronized with loading of the input pattern **X**-vector, from software, to interface BRAM and then into the two shift-registers (one for parallel multiplication and the other for serial one). The loading of **X**-vector is performed by the secondary state machine SM1. While the primary state machine handles the loading of weight matrices into the Static Random Access Memory, SRAMs and also the parallel load of BRAMs. After loading weight matrices into SRAMs, during the parallel load of a weight matrix into the BRAMs, the primary state machine sends a *start* signal and the *index* value of the weight matrix being loaded to the secondary state machine SM1. The start signal triggers the work of SM1. Once the SM1 finishes its task it sends a *done* signal to the primary state machine. After the parallel load of a weight matrix is done the primary state machine performs 10 recall operations. The recalled output **X**-vector is then sent to the software on the host PC. This task is controlled by a secondary state machine SM2 that receives a start *signal* and *index* value from the primary state machine. It sends a *done* signal once it has completed its task. Except for the first BSB model, the sending of the output **X**-vector operation to the software is done also in parallel with the loading of the weight matrix (i.e., the two operations sending the output **X**-vector to the software and loading the next input **X**-vector for next recall operation from the software are overlapped with the loading of the weight matrix into the BRAMs). The

final BSB recall only sends the output **X**-vector to the software. The entire operation flow is shown in Figure 15. For the current experimental runs, the weight matrices and the input **X**-vectors are read in from separate files and then stored in an array in the main host code. The recalled **X**-vector is stored one by one in an array and then all the 256 of them are stored to a file.

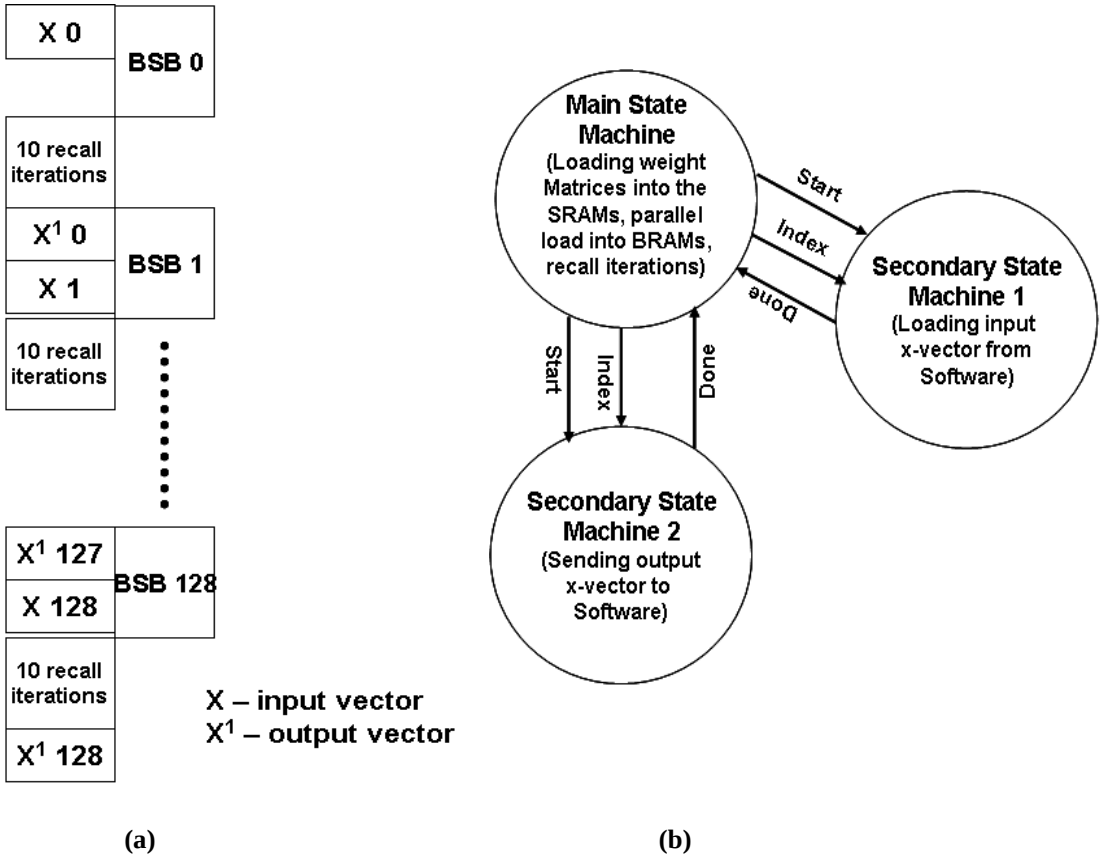


Figure 15. (a) BSB hardware operation flow and (b) interactions among state machines.

3.3. Integrating pub/sub protocol with BSB hardware/software

With the resources currently available to us, we designed a simple communication scenario to integrate the hardware-accelerated BSB application with the pub/sub protocol.

As shown in Figure 16, the BSB design is implemented on workstation B. The pub/sub protocol software is run on both A and B individually. The pub/sub software consists of an event system router and the application programs specified by users. The event system service is a router in the same way a Cisco router is on a physical network. It is inherently bi-directional, so publishing and subscribing applications can be run on both workstations. BSB inputs "*x_vector*" and the recalled outputs "*x_recalled*" are published with different event types: *event1* and *event2*.

A subscribes to date with "*event2*" type, and B subscribes to "*event1*". Therefore when A publishes the *x_vector*, B will detect the *event1*-type-message it is listening for, and retrieves the *x_vector* from the network. Similarly, when B finishes computing the recalled vector and publishes the *x_recalled*, A will retrieve the message.

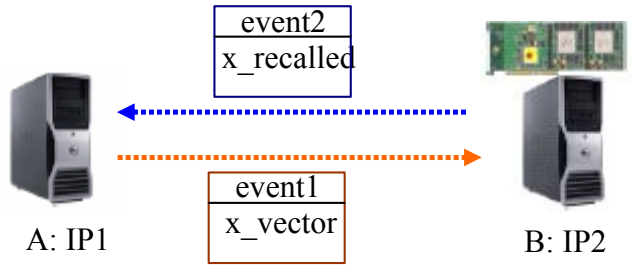


Figure 16. A simple communication scenario.

3.4. Implementation of cogent confabulation algorithms on Cell processor

The training software runs in the client server mode on the IBM Cell processor [9]. The 6 SPEs are clients that perform the training function. During the process, they issues service requests to the PowerPC Processing Element, PPE. The PPE works as a server that collects and responds to those requests. The SPE sends the request to the PPE through the output mailbox and receives the feedback from PPE from the input mailbox.

Figure 17 gives the overall flow of the training function that runs on an SPE. The block diagram is color coded. Different colors are used to represent different tasks, which include mailbox read/write, DMA read/write, normal operation and sub-functions.

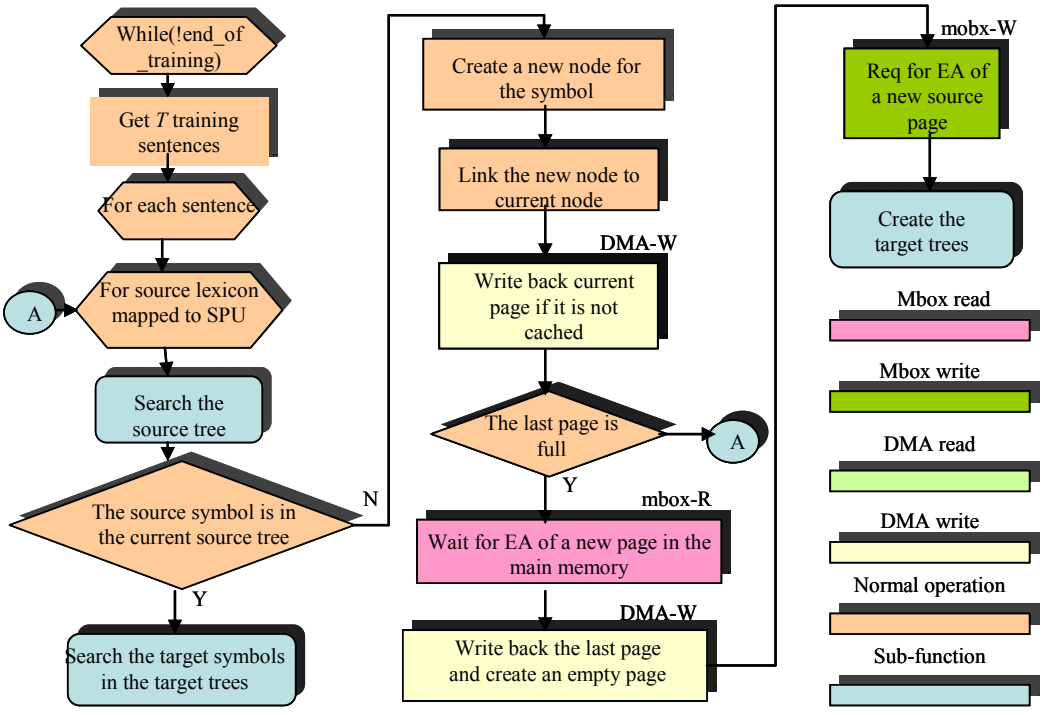


Figure 17. Flow diagram of the training function that runs on the SPE.

While the training is not ended, the SPE will get *T* training sentences from the designated area in the main memory. To improve the Direct Memory Access, DMA efficiency, *T* is set to 64. For each training sentence, and for each source lexicon that is mapped to it, the SPE first searches the source list to look for the source symbol. If the source symbol is in the list, then the SPE go on searching for the target symbols in the target list. If the source symbol is not found, then a new node is created for the symbol and the node is added to the last page of the source list. The SPE fills the current node's child information with the page id and page offset of the new node. Since the current node has been modified, if it is not the last page or one of the first *L* pages, then it will be written back. If the last page is not full, then the SPE will continue processing the next source lexicon, otherwise, it must write the last page back to the main memory. To write back the last page, a page must first be allocated in the main memory and its address must be sent to the SPE. This task is performed by the PPE. The SPE waits until it receives the Effective Address, EA of a new page from the PPE. It will then write back the last page to this address and at the same time create an empty page in the local storage. The page that has been deposited to the main memory must be linked to the end of the source list and again this job is performed by the PPE. Finally, a new target page must be created to store all the target nodes that are linked to the source node.

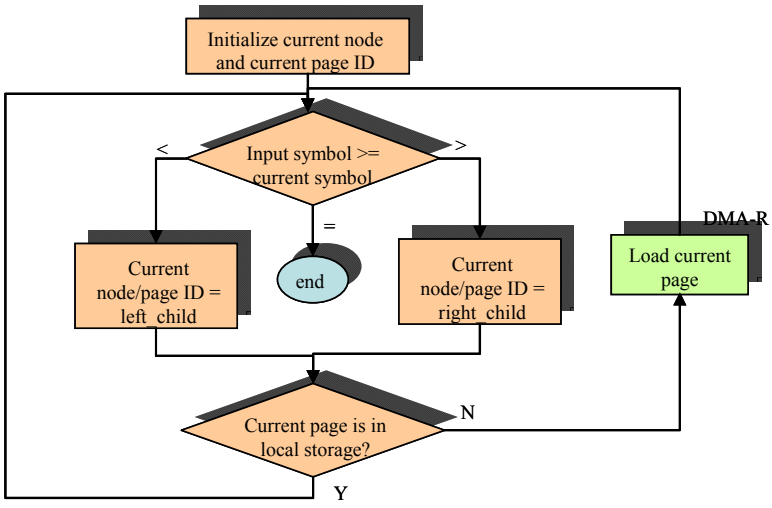


Figure 18. Flow diagram of the source list search function.

Figure 18 shows the flow diagram of the source list search function. At the beginning of the function, the current node is initialized to the first node in the first page. If the input symbol is equal to the symbol that is stored in the current node, then the program ends, otherwise, if the input symbol is greater/less than the current symbol, then its right/left child will be visited. As we mentioned in the previous section, a child node is identified by its page index and offset. If the child page is already in the local storage then we set the child node as the current node and continue the search, otherwise we will load the child page to the local storage and continue the process.

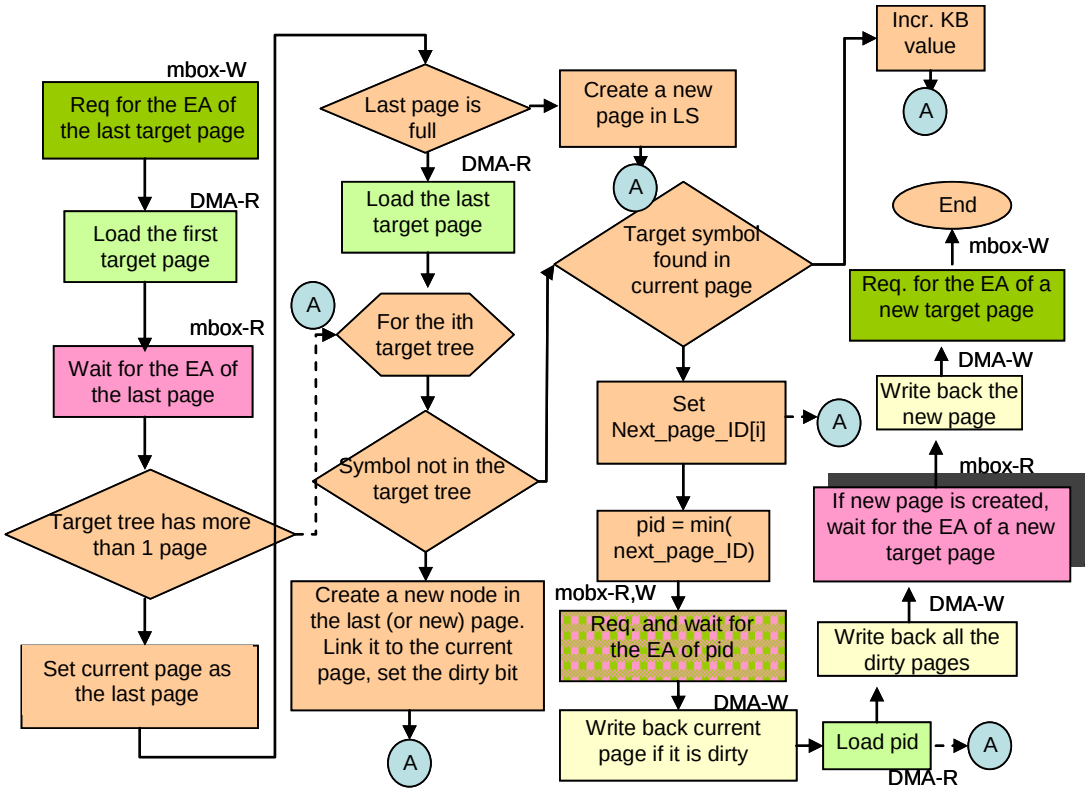


Figure 19. Target list search.

Figure 19 gives the detailed information about target list search process. When a source node is visited, the SPE retrieves the address of the first target page that is linked to it. However, it does not have the information of the address of the last page. The SPE sends a request to the PPE for the effective address of the last page. The PPE searches for the last page alone in the link list and sends back its address. At the same time, it also informs the SPE if the target list has only one page or the last target page is full. If the target list has more than one page and if the last page is not full, the last page will be loaded into the local storage. If the last page is full, then a new page is created in the local storage and it will be considered as the last page of the target list. For each target tree that is connected to the source node, the SPE will set the current search node to the *i*th node in the first page, where *i* is the index of the target tree. If the input symbol is in the current node, then the KB value of the current node is incremented, otherwise, the page ID of the child node is set to the *i*th entry of the next_page_ID array and the SPE goes on processing the next target tree. After all target trees have been processed, a set of next_page_IDs are collected. The minimum one is chosen and loaded from the main memory to replace the current page in the local storage. Before this happens, if the current page is dirty, it must be written back. After the child page is loaded, the SPE will repeat the previous procedure until all symbols have been found. If a symbol cannot be found in the target list, a new node is created for it. If the last page is not full, the new node will be inserted into the last page, otherwise it will be inserted into the new page. Finally, if the last page or the new page is dirty, then they will be written back.

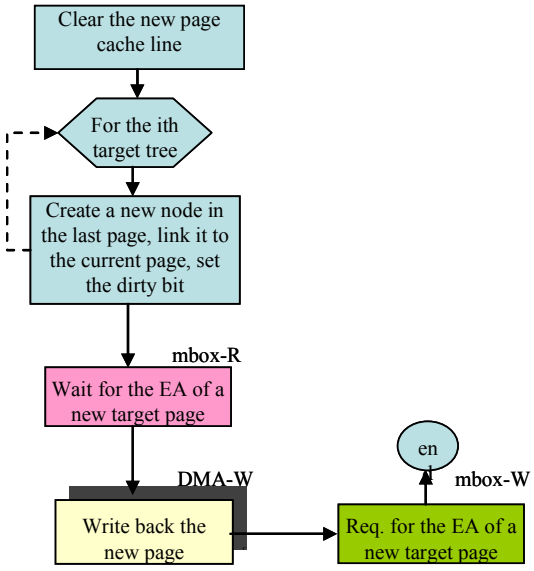


Figure 20 Flow of creating a target list

Figure 20 shows the flow of creating a new target list. The SPE will initialize a new page. For each symbol in the target lexicon that is linked to the source lexicon, a node is created in the page. The SPE waits for the PPE to allocate a page in the main memory and then use DMA write to write back the new page.

The PPE continuously checks the output mail box for requests from each SPE. After processing these requests, the results are sent to the input mail box of each SPE. Overall, there are 5 different requests:

1. Request for a new target page. Before the SPE writes back a new target page, the PPE needs to allocate a space for the page in the main memory.
2. Request for a new source page. Before the SPE writes back a new source page, the PPE needs to allocate a space for the page in the main memory.
3. Request to load the next target page. During the search for target symbol, if a page needs to be loaded, the PPE will send the page ID to the SPE and the SPE will search through the link list to find the address of the requested page.
4. Request to load the last target page. When a source node is visited, the SPE gets the address of the first page in the target list. It will then request the address of the last page in the list.
5. Request to link the new source page. The SPE informs the PPE that a new source page has been written back and the PPE will link it to the end of the source list.

4. RESULTS AND DISCUSSION

4.1. Pattern recognition application using BSB hardware

Three different types of patterns were trained and recalled using the 128-neuron BSB model. They are line patterns, alphabet patterns and symbol patterns

In each of the above patterns the first 28 bits represent the *Tag* while the remaining 100 bits represent the 10-by-10 black-and-white image of the pattern, but the way by which tag is represented differs with the type of the pattern being implemented. For example, for line patterns Bit 3, Bit 4, Bit 5 and Bit 6 are used to differentiate the type of pattern being trained or recalled, while the remaining bits 0-2 & 7-28 are always -1s. There are four different Line patterns, they are *vertical*, *horizontal*, *forward slash* and *backward slash*. The tag bit representation for the patterns is shown below

Table 1. Flag bits for different line patterns.

Type of pattern	**Bit3**	**Bit4**	**Bit5**	**Bit6**
Vertical	+1	-1	-1	-1
Horizontal	-1	+1	-1	-1
Forward slash	-1	-1	+1	-1
Backward slash	-1	-1	-1	+1

While the remaining 100 bits represent a 10 by 10 pattern. The '+1' represents the *Energy pixel* while the '-1' represents the *No Energy pixel*. Figure 21 shows a vertical line pattern and its representation.

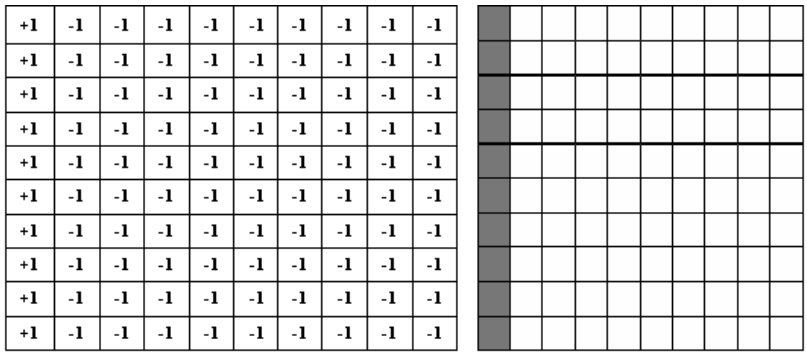
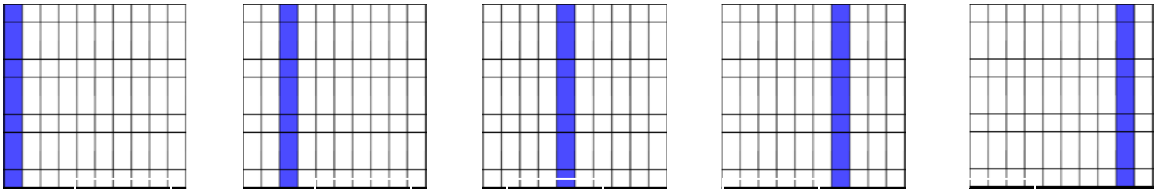


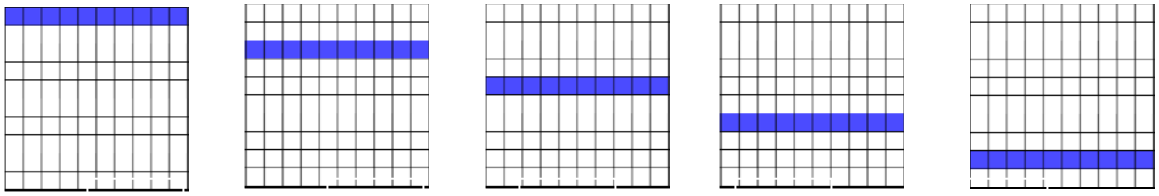
Figure 21. Vertical line pattern and its representation.

Total of 20 Line patterns were trained, five each of vertical, horizontal, forward slash and backward slash. The complete training set is show in Figure 22.

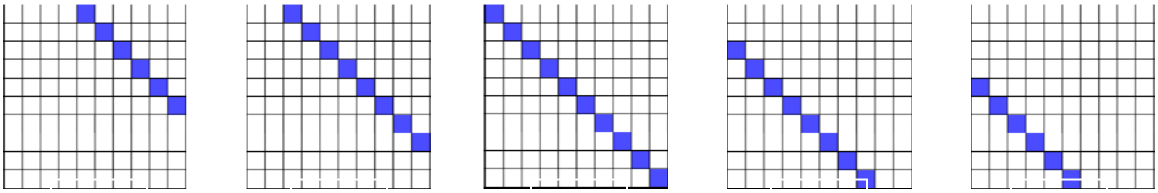
Vertical patterns:



Horizontal patterns:



Backward Slash:



Forward Slash:

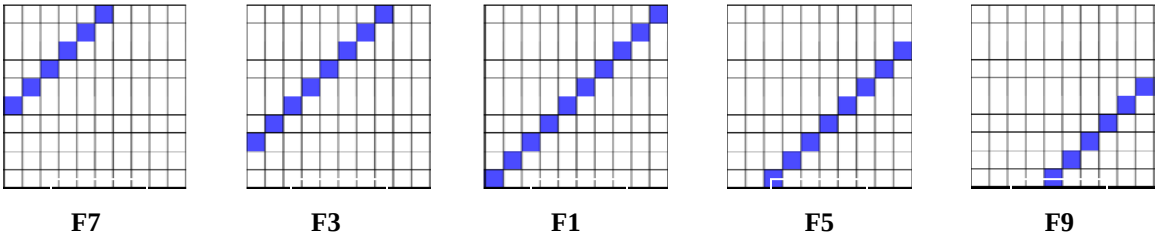


Figure 22. Complete training set of line patterns.

The above patterns were recalled successfully by our design. In addition, their variations, such as the thicker line patterns, V12 (first and second columns in V1 pattern are black), V34, V56, V78, V910, H12, H34, H56, H78, H910, F12, F34, F56, F78, F910, B12, B34, B56, B78 and B910 were also recalled successfully. All the patterns without tags (tag bits were placed with zeroes) were recalled successfully.

For alphabet and symbol patterns the tag bit representation was different. One hot encoding scheme was employed to represent the tag for each letter. (For example, the letter A had the first bit as +1 while the remaining bits were -1, letter B had second bit as +1 while the remaining were -1, etc.). The same one hot encoding scheme was used for symbol patterns. Each symbol had a unique tag. The different letter and symbol patterns are shown in Figure 23.

Figure 23. Complete training set of alphabetic and symbol patterns.

All the above patterns were recalled successfully without tags. Recall was attempted on some of the above patterns with tag and partial input. Some of the partial patterns converged successfully to the original patterns, while others failed to converge. For example, for the letter A, the original pattern was trained for 500 iterations along with other letters. The original pattern without tag took 9 recalls to converge. Different partial patterns (shown in Figure 24) were recalled without tags. Some of the patterns were able to converge successfully, while a few failed to converge to the original pattern.

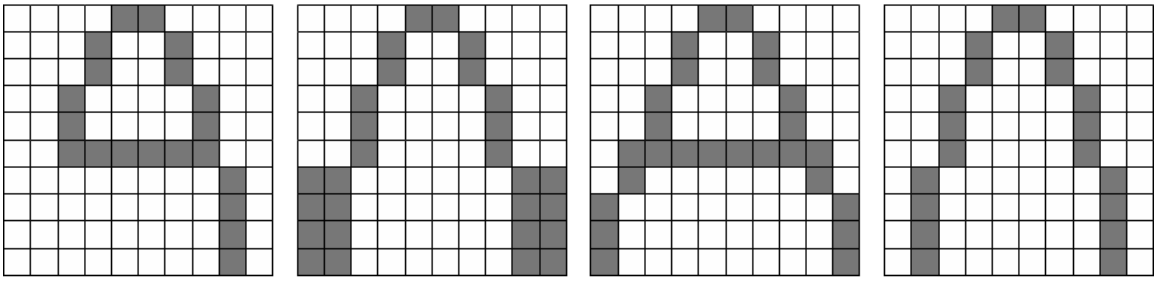


Figure 24. Partial patterns of letter "A".

The pattern in image (a) took 20 recall iterations to converge successfully to the original pattern while patterns (b) and (c) took 27 and 41 recalls, respectively to converge correctly. But the pattern (d) failed to converge to the original pattern after 16 recalls.

Table 2. Statistics of successful and unsuccessful recall operations.

	# of recalls before successful converge	# of recalls before unsuccessful converge
Exact training pattern without tag	8 ~ 11	N/A
Similar/partial pattern without tag	11 ~ 41	16 ~ 40

The platform for running the BSB recalls is the Annapolis Wildstar II Pro PCI-X card with one Xilinx Virtex II Pro XC2VP70 FPGA and six Samsung QDRII SRAMs. The system clock frequency is 100 MHz.

64 different Input patterns (32 line patterns and 32 alphabet patterns) replicated four times to form 256 Input patterns were inputted to the 256 128 neuron BSB model through the file "x_vector256.txt". The corresponding weight matrices of the Line Patterns and that of the Alphabet Patterns were also input using two files "weight_matrix.txt" and "weight_alpha.txt". For each of the input patterns 30 recall iterations were performed and then output the recalled pattern to the software where they are stored in an array one by one. At the end of recall of the 256^{th} pattern all recalled patterns were sent to a file "x_recalled.txt". The file "x_input.txt" had all the 64 different Input patterns represented in 0's & 1's. In the output file "x_recalled.txt" all the recalled patterns were represented in 0's & 1's (where -32767 &

+32767 were represented as 0 & 1 respectively). If in 30 recall iterations any of the input bits did not converge to either -32767 or +32767 then it was represented as 'N'. The screenshot of the output file "x_recalled.txt" is show below in Figure 25. The output recalled pattern is shown below its respective input pattern. The output pattern is represented in Tag and Pattern format. We can see on the right half side of the screenshot that the "Partial A" input pattern is not fully recalled at the end of 30 recall iterations. The input bits that did not converge are marked as 'N'.

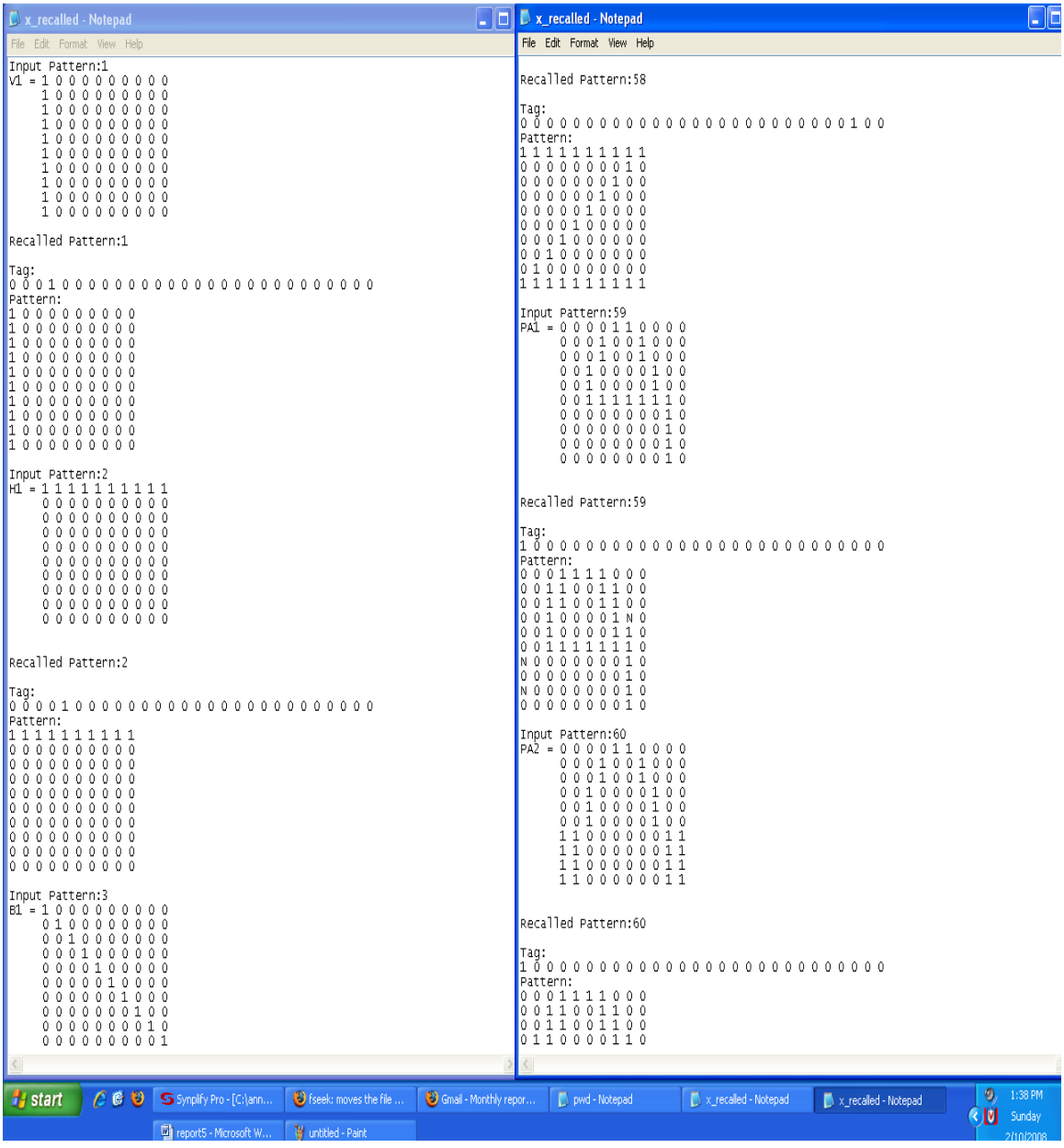


Figure 25. Screenshots of the recall output files.

4.2. Performance characterization of cogent confabulation algorithms on Cell

Five different training files were tested to evaluate the performance of the cell program. They are:

A. random-400: randomly generated file (400 sentences)

B. random-600: randomly generated file (600 sentences)

C. story-short: children's short story (126 sentences)

D. story-long: children's long story (1092 sentences)

E. science: part of a science paper about relativity (334 sentences)

The size of the source and target page is 64 nodes, i.e. N=M=64. For each source tree, 11 source pages are cached in the local storage, i.e. L = 9.

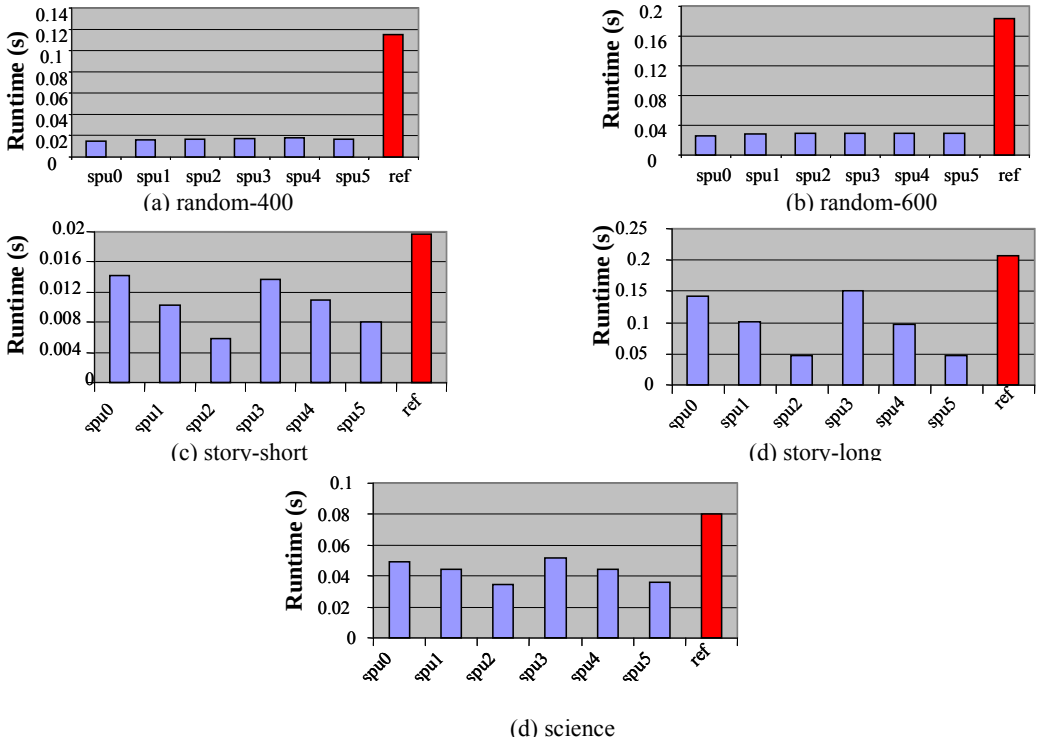


Figure 26. Performance comparison.

Figure 26 (a)~(d) gives the performance comparison between the cell program and single processor program for those 5 test cases. Table 3 gives the performance information of the cell program, including its speedups over the single processor program, the idle-to-busy-ratio of the PPE for each test case and the parallelism of the program when running different test cases. The speedups are calculated as the training time on cell divided by the training time on single processor. The parallelism is calculated as the sequential runtime divided by the parallel runtime.

Table 3. Cell program performance information.

Test cases	Random-400	Random-600	Story-short	Story-long	Science
Speedups	6.25	5.88	1.36	1.37	1.56
Idle-to-busy-ratio	1:2.3	1:2.4	1:15	1:72	1:35
Parallelism	2.8	2.6	3.0	1.6	2.1

The results show that the cell program provides less speedup for the three real life test cases. A future software profile shows that these test cases have low PPE idle-to-busy-ratio, which is mainly caused by large number of last target page requests issued by the SPE.

Table 4 shows the detailed profile information of the SPE-PPE requests.

Table 4. Software profile of SPE-PPE requests.

Test cases	New source page	Next target page	Last target page	Link source page
Random-400	280	0	226	280
Random-600	400	0	498	400
Story-short	54	0	839	54
Story-long	226	229	11287	226
Science	118	1	439	118

We further optimized the cell program so that each SPE handles the last_target_page_request locally. This reduces about 60% of the runtime. Figure 27 shows the performance of the optimized software and

Table 5 gives the speedups of the optimized cell program over the single processor program.

Table 5. Speedups of the optimized cell program.

Test cases	Random-400	Random-600	Story-short	Story-long	Science
Speedups	8.3	9.6	3.0	3.9	4.0

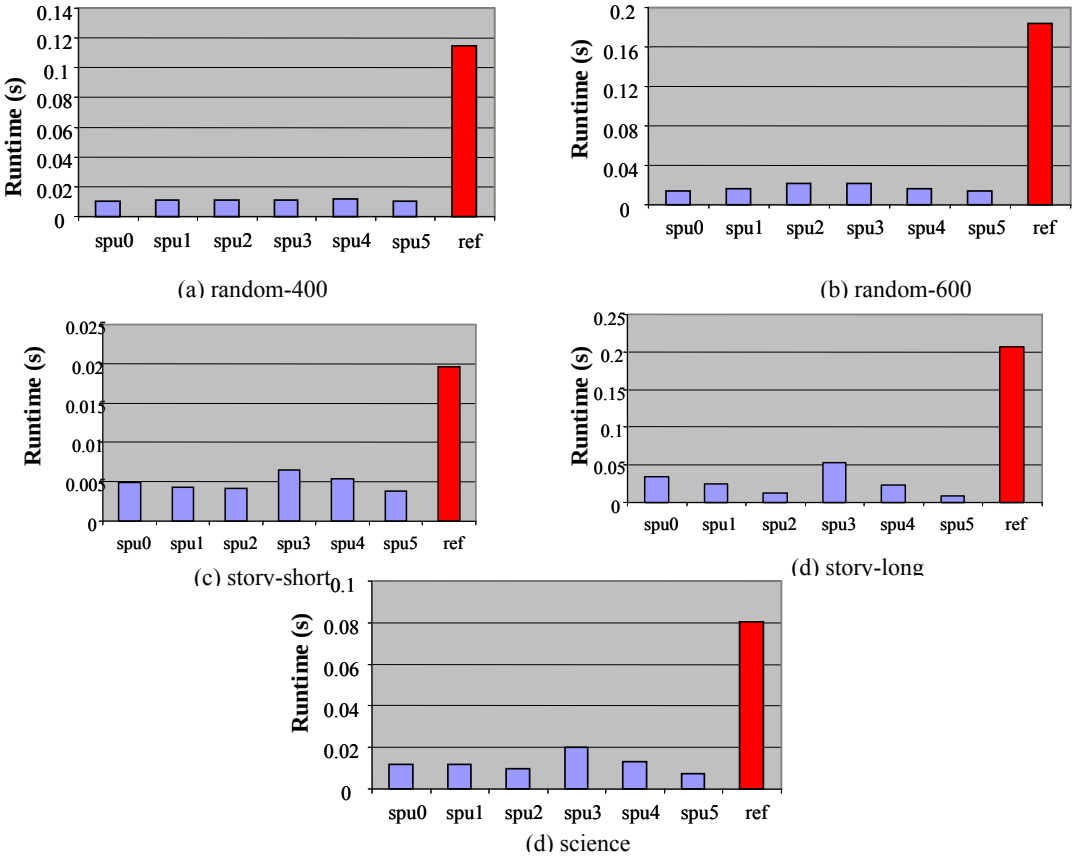


Figure 27. Performance of optimized software.

5. CONCLUSIONS

We have described the working hardware and software developed to realize large-scale Brain-State-in-a-Box (BSB) models on a workstation with hardware acceleration using a Field Programmable Gate Array (FPGA), as well as the implementation of confabulation based knowledge base training function on the Cell Broadband Engine (CBE). Actual runtime measurements show that by applying the hardware and software optimization techniques developed in this research, we were able to significantly improve the performance of both algorithms, comparing to the normal software solutions on general-purpose processors.

We are honored to be part of, and be able to contribute to the cognitive computing research efforts at AFRL Rome Site. We would like to thank Dr. Richard Linderman, Dr. Tom Renz, Mr. Daniel Burns, Mr. Michael Moore, and many other researchers for their strong support and valuable advises during the course of this project.

6. REFERENCES

[1] "*Associative Neural Memories: Theory and Implementation*," Mohamad H. Hassoun, Editor, Oxford University Press, 1993.

[2] "*On Intelligence*," Jeff Hawkins, Sandra Blakeslee, Times Books, Henry Holt and Company, LLC, 2004.

[3] "*WILDSTAR II for PCI Data Sheet*," Annapolis Micro Systems, Inc.

[4] "*Virtex-II Family Product Tabl*e," Xilinx, Inc.

[5] "*Virtex-II Pro Family Product Table*," Xilinx, Inc.

[6] Patrick Eugster, Pascal Felber, Rachid Guerraoui, and Anne-Marie Kermarrec, "The Many Faces of Publish/Subscribe," *ACM computing Surveys*, 35(2), June 2003.

[7] R. Hecht-Nielsen, "Mechanization of Cognition", *Biomimetics, CRC Press*, 57–128.

[8] Q. Qiu, D. Burns, M. Moore, R. Linderman, T. Renz, Q. Wu, "Accelerating cogent confabulation: An exploration in the architecture design space," *IEEE World Congress on Computational Intelligence*, June 2008.

[9] M. Kistler, M. Perrone, F. Petrini, "Cell Multiprocessor Communication Network Built for Speed," *IEEE Micro*, 2006.